

Wub Unofficial Bootstrap Tutorial [WUB Tutorial]

Wub 6.0

Part 1



Alie Tormusa Koroma [ATK]
[akoroma at consultant dot com](mailto:akoroma@consultant.com)

01 October 2015

Introduction

I was never a web developer and an intermediate coder I'll call myself. But when it became inevitable that I did or learn web development? I chose the latter and I was not going to learn a new language doing so.

Thank goodness Tcl provides a few options but Wub is my favourite because of its framework approach, not like I'm in any place to comment on the others nor have I attempted to use any other but TclHttpd. I found it very difficult for a newbie with no prior web development experience to grasp the Wub concept. I struggled the same way with TclHttpd before switching to Wub.

As I struggled with what might be simple for most people, I thought there might be others like me that will benefit from the little understanding I had built of the product. Wub is my favourite Tcl (**T**ool **c**ommand **l**anguage) tool to say the least and will like for others to rip its benefits.

Wub as a framework does not provide a graphical user interface with point-and-click or drag-and-drop functionalities. You might need some bootstrapping to leverage its power. I hope this document will provide you with the understanding required to do so.

Getting Started

Wub works out of the box, the following steps are from the quick start doc that comes with the software when you download it;

- Step 1: Download the code (which you've probably done already)

```
svn checkout http://wub.googlecode.com/svn/trunk/ wub
```

- Step 2: Change into the Wub directory:

```
cd wub
```

- Step 3: Run the demo application:

```
tclsh8.6 Wub.tcl
```

- Step 4: Connect and Go.
the server should now be listening on port 8080. Try it (<http://localhost:8080>).
- Step 5: For later

```
edit site.config to your taste
```

Some start-up texts are displayed on your terminal which will only makes sense as you gain more understanding. We will get to that later on in the document. What actually happens at this stage is, Wub reads its configuration file (`/${install_dir}/site.config`) and performs all the configured actions. This means you can direct the web server to do things like...if you receive a request for url **http://x**, call code **proc handle_x** and Wub will pass all the browser query key-pair values to your code **handle_x** as arguments. This is called a direct domain and it's a ways of channelling a HTTP client request to your handling code (namespace or class).

Let's say your web site address is www.mydomain.co.za and the web server powering this site is running Wub. If someone types that url on their browser, the request will travel over the internet using name resolution (DNS) all the way to your machine or server running Wub on port 8080 or 80. We are only interested in when that communication reaches the Wub server for the scope of this document.

Wub will be listening for this request on the configured port number, the default is 8080. this is depicted in this section of *site.config* as follows;

```
# Listener module defines the HTTP network listener
# connections are established through this module, and simply passed
# into the protocol engine. This module allows you to define where
# the server should accept connections (which port, which interface, etc.)
Listener {
    -port 8080          ;# HTTP listener's port
    #-myaddr X         ;# make listener listen on the nominated interface
}
```

If you change this port number to 80 under Ubuntu, you can only start-up the web server using sudo (only tested on Ubutu). Wub listens on all interfaces by default, comment out -myaddr and add a interface name to make listener listen on the nominated interface.

A web server serves content using the HTTP protocol. The content can be static or dynamic. To server dynamic content, you will manipulate the request and respond with something relevant to the user's request. For example if a user fills a form at **www.mydomain.co.za/register** and clicks submit, the browser will pass the url plus key-pair values from the form to the web server. This url and key-pair value together is called a *query string*. It will look as follows;

```
http://www.mydomain.co.za/register?firstname=Tormusa&lastname=Koroma
```

The query string may not always look as above, it may contain encoded non-ascii characters. If a procedure by the name register exists within the namespace or class configured for this domain (direct Domain), Wub will call it with the arguments values of *firstname* and *lastname* (*Tormusa and Koroma*). This means your proc should look as follows;

```
proc /register {firstname lastname} { . . . }
```

If such proc does not exist in your program or namespace, it will be passed to the root proc of your namespace or class;

```
proc / {r args} { . . . }
```

Once the request reaches your logic, you are free to manipulate its values and respond meaningfully. For example, you can validate and authenticate the *firstname* and *lastname* values and if a user with such first and last names exists in your database, you can grant the user access (login) and return his/her alias name.

In other web development environments like apache with php for example, the server and client parts are cleanly distinguished by `<html>` and `<%php>` tags. With Wub, everything is Tcl and it's difficult to make such reference here using Direct Domains. The server is Tcl and the scripting language is Tcl so there's no need to distinguish Tcl from Tcl. For that reason, in Wub you just write code using Tcl and Wub commands without separation. So while transitioning from your old habits into using Wub, always remember... *"everything is Tcl"*.

Utilities and Domains

A framework from my definition is a skeletal foundation for developing upon. This development will be governed by the framework's guidelines or rules which are implemented in its capabilities and tools. Wub is not only a HTTP server but a framework for building HTTP applications. It provides the tools to assist you in rapidly doing so (Utilities and Domains).

All the utilities are readily available to you under `/$install_dir/Utilities`. They are used without the need to **package require** them first. Examples of such tools are `Http`, `Html`, `Form`, `Cookies` to name a few. Domains on the other hand needs to be configured in `site.config` before use. They reside in the `/$install_dir/Domains` directory (`jQuery`, `Nub`, `Direct` etc). We will visit a few of them in a minute.

With all these available utilities and tools provided by the framework, building a web page in Wub can be very quick and simple. The process involves constructing the page (html), apply functionality and styling (`jquery` & `css`)...then return the content back to the browser using `Http` utility.

Here is a simple example (excuse the html tags).

```
# build the page content in variable mypage
set mypage `
    <html>
        <head>
            <title>My Wub Page</title>
        </head>
        <body>Welcome to my first Wub page</body>
    </html>
`
# apply style called mycss to the page
set r [Html style $r mycss]

# return the content to browser
return [Http Ok $r $mypage]
```

You can neatly embed your CSS within your code as opposed to having it in an external file. This is an excellent feature provided by Wub and we will get into the specifics a little later on.

Configuring Wub for your site

There is only one configuration file called **site.config** and it resides under `/$install_dir`, all configuration changes going forth will mean editing this file. We will configure a `Direct` domain for our site that will tell Wub to dispatch all URL requests to our specified namespace or class. This concept helps you as a developer maintain an organized code in the sense that your whole web site is encapsulated in a single namespace/class.

Add the following entries to create and execute the namespace **MySite** for requests to **/mysite** (this will mean the user typed `www.mydomain.co.za/mysite`). The root proc of namespace `MySite` is called (`proc / {}{...}`).

We will now configure a `Direct` domain for our site;

Add a domain `/mysite/` of type Direct for `www.mydomain.co.za/mysite` requests that executes namespace `MySite`

```
/mysite/ {
    domain Direct
    url /mysite
    namespace ::MySite
}
```

Change the root URL redirect to domain `/mysite/`. The out of the box configuration will redirect to `/wub`.

```
# the URL / redirects to /wub
/ {
    #redirect /wub/      ;# this was the default for the wub page
    redirect /mysite/
}
```

On your browser, type <http://localhost:8080>

The browser address will automatically change <http://localhost:8080/mysite/> this is because of the configuration containing the line `redirect /mysite/`

Wub server needs to be told where to send the user request when it receives it. This way, you can build logic to handle the request and send back a response. Converting requests to responses in Wub simply means the developer will modify the request dictionary and return it as the response.

Our `proc / {r args} { . . . }` in namespace `MySite` will receive the request dict (`r`) for our Direct Domain request. In the Prac that follows, we simply print out the variable `r` which contains the request dict.

Your namespace code should reside in a file called `local.tcl` under the `$install_dir`. This file is sourced during server start-up. The complete code will look something like the following;

Prac 1.0 sample web page code in `/$install_dir/local.tcl` file

```
namespace eval ::MySite {

    proc / {r args} {
        puts "root::MySite:RequestDict_r: -->\n $r"
        set content $r

        return [Http Ok $r $content text/html]
    }

    namespace export -clear *
    namespace ensemble create -subcommands {}

}
```

Figure 1.0 an example request dict dump

```
root::MySite:RequestDict_r: -->
-host localhost -port 8080 -httpd {::Httpd new} -id ::oo::Obj51 -server {127.0.0.1 127.0.0.1
8080} -scheme http -sock sock2623a90 -cid ::oo::Obj57 -ipaddr 127.0.0.1 -rport 57504
-received_seconds 1443093470 -server_id {Wub 6.0} -pipeline 3 -send ::Httpd::coros::obj57
-transaction 3 -time {connected 1443093470010508} -header {GET /mysite/%2F HTTP/1.1} -method
GET host localhost:8080 -clientheaders {host user-agent accept accept-language accept-encoding
referer connection} user-agent {Mozilla/5.0 (X11; Ubuntu; Linux x86 64; rv:31.0)}
```

```

Gecko/20100101 Firefox/31.0} accept
{text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8} accept-language {en-ZA,en-
GB;q=0.8,en-US;q=0.5,en;q=0.3} accept-encoding {gzip, deflate} referer
http://localhost:8080/mysite/logout connection keep-alive -version 1.1 -ua {ua {Mozilla/5.0
(X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0} id FF version {}
mozilla_version 5.0 extensions {} platform X11 security Ubuntu subplatform {Linux x86_64}
language rv:31.0 product {Gecko 20100101 Firefox 31.0}} -ua_class unknown -normalized 1
-path /mysite/ -url http://localhost:8080/mysite/ -uri http://localhost:8080/mysite/ -forwards
{} -encoding binary -received 1443093471248483 -cookies {} -Query {} -prefix /mysite/ -suffix
{} -extension {} content-type x-text/html-fragment -dynamic 1 -extra {} -fprefix / -cprefix /

```

The `Http` utility is used for generating HTTP responses (Ok/200, ServerError/500, NotFound/404, Forbidden/403 etc) and the package is located in `/$install_dir/Utilities`.

```
Http Ok $r $mypage text/html
```

The above code generates a 200/Ok response with the dictionary `$r` and page content `$mypage` of content type `text/html`.

The request dictionary holds valuable information like the query string for example. This made room for me to make mention of another useful tool called Query. This query string for instance, is contained in a dict key called **-Query** (highlighted in the figure 1.3). It contains no values (**-Query {}**). The `Query` command is used to manipulate the content of **-Query** in the request dictionary.

One can equally use the `ncgi` package that comes with **Tcllib** to manipulate this query string. However, I prefer to use the `Query` utility when using the framework.

Let's develop our page a little more by adding some html and changing the content to something plain and simple.

Prac 1.1 building the web page

```

namespace eval ::MySite {
    proc / {r args} {
        puts "root::MySite:RequestDict_r: -->\n $r"

        set mypage {
            <html>
                <head>
                    <title>WUB Tutorial</title>
                </head>
                <body>
                    <font size=18px><b>WUB Tutorial</b></font><br><hr
                    color="blue"><br><br>
                    Welcome to my first Wub page
                </body>
            </html>
        }

        return [Http Ok $r $mypage]
    }

    namespace export -clear *
    namespace ensemble create -subcommands {}
}

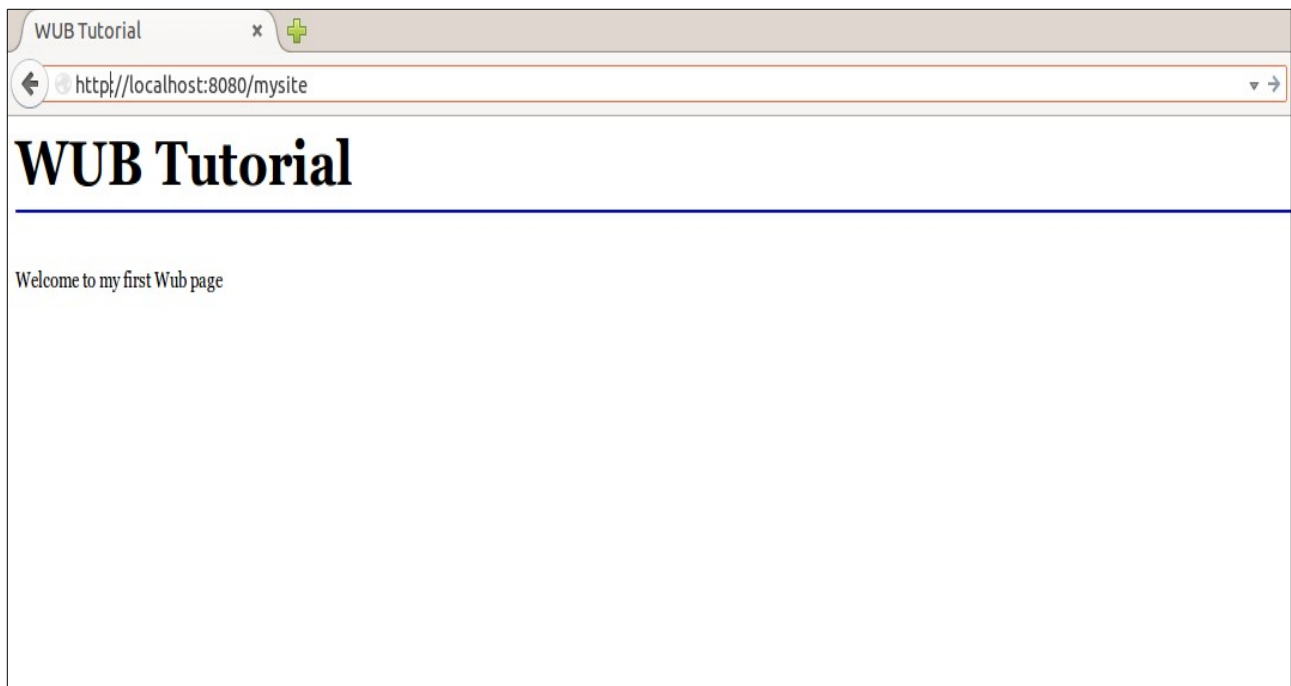
```

Note with the `Http` command this time, we did not add the content type. The default is `text/html`.

If you type <http://localhost:8080> in your browser, your page should look like the image

below;

Figure 1.2 *welcome to my first Wub page*



Using Query

“Select fname, lname, startdate from students;”

What does a select query have to do with Wub? Well...not much, but I want to draw from the word *query* as a way of formally introducing Wub's `query` utility. It is such an important utility I feel it deserves the wow. We will view the query string at the Wub server to better understand how you use the `query` utility.

Change your browser URL to be <http://localhost:8080/?fname=Alie&lname=Koroma&startdate='09-SEP-2015'> and press enter.

If we take a look at the request dict, two elements of interest will be the **-query** and **-Query**. They are the query string and the **-Query** key-pair (for lack of better name) respectively.

Excerpt of -query and -Query from the request dict as posted by your browser to Wub web server

```
-query fname=Alie&lname=Koroma&startdate=%2709-SEP-2015%27
-Query {fname {Alie {-count 1}} lname {Koroma {-count 2}} startdate {'09-SEP-2015' {-count 3}}}
```

Prac 1.2 modify our page to capture or extract the query string values and use them.

```
namespace eval ::MySite {
    proc / {r args} {
        puts "root::MySite:RequestDict_r: -->\n $r"
        set mypage {
            <html>
                <head>
                    <title>WUB Tutorial</title>
                </head>
                <body>
                    <font size=18px><b>WUB Tutorial</b></font><br><hr color="light-
                    blue"><br><br>
                }
                # extract the values from the query string or the request dict
                set fName [Query::value [Query::parse $r] fname]
                set lName [Query::value [Query::parse $r] lname]
                set sDate [Query::value [Query::parse $r] startdate]

                append mypage "Welcome $fName $lName<br>"
                append mypage "This is Wub Unofficial Bootstrap Tutorial.<br>Start Date: $sDate"

                append mypage "</body></html>"

                return [Http Ok $r $mypage]
            }
        namespace export -clear *
        namespace ensemble create -subcommands {}
    }
}
```

Figure 1.2 shows our response to that browser request. The *fname*, *lname* and *startdate* values from the query string are used in the response.

The `Query::value` command takes two arguments, query and element.

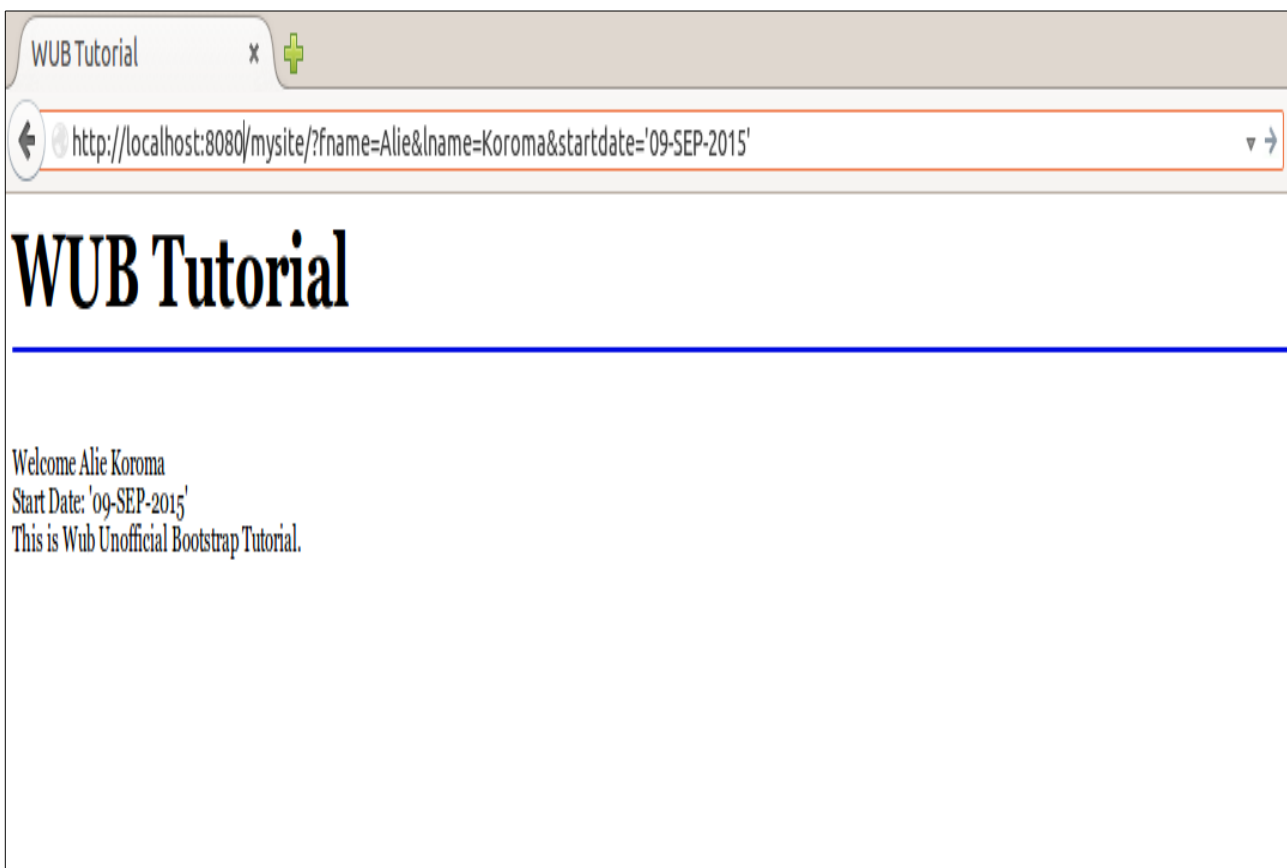
```
Query::value [Query::parse $r] fname
```

arg1 is the query string in the format returned by `Query::parse` command
arg2 is the exact name of the query string element that must exist (e.g fname)

`Query::parse $r` returns the query string in the form of -Query dict element. Below is the actual output;

```
fname {Alie {-count 1}} lname {Koroma {-count 2}} startdate {'09-SEP-2015' {-count 3}}
```

Figure 1.2 using values from a query string



Referencing a query element that doesn't exist will cause an error. Use the `Query::exists` to check its existence before referencing its value where necessary. It returns a 1 or 0 for true or false respectively.

The following command will return 1 if element fname exists in query string returned by `Query::parse` command;

```
Query::exists [Query::parse $r] fname
```

Prac 1.3 modifying `proc` / demonstrating the use of `exists` in order to enhance the logic.

```
namespace eval ::MySite {
    proc / {r args} {
        puts "root::MySite:RequestDict_r: -->\n $r"
        set mypage {
            <html>
                <head>
                    <title>WUB Tutorial</title>
                </head>
                <body>
                    <font size=14px><b>WUB Tutorial</b></font><br><hr color="light-
                    blue"><br><br>
                }

            if { [Query::exists [Query::parse $r] fname] } {

                # extract the values from the query string or the request dict
                set fName [Query::value [Query::parse $r] fname]
                set lName [Query::value [Query::parse $r] lname]
                set sDate [Query::value [Query::parse $r] startdate]

                append mypage "Welcome $fName $lName<br>Start Date: $sDate<br><br>"
                append mypage "This is Wub Unofficial Bootstrap Tutorial.<br><br>"

            } else {

                set encStr [Query::encode "?fname=Alie&lname=Koroma&startdate=[clock
                format [clock seconds] -format {%d-%b-%Y}]" ]
                append mypage "My login bookmark: <a href=$encStr>Auto Login</a>"
            }
            append mypage "</body></html>"

            return [Http Ok $r $mypage]
        }

        namespace export -clear *
        namespace ensemble create -subcommands {}
    }
}
```

The modification checks if the *fname* element exists and extracts the values of *fname*, *lname* and *startdate* from the query string. Otherwise, it constructs and encodes a query string and uses it in a link.

```
set encStr [Query::encode "?fname=Alie&lname=Koroma&startdate=[clock format [clock seconds]
-format {%d-%b-%Y}]" ]
append mypage "My login bookmark: <a href=$encStr>Auto Login</a>"
```

In the above line, we create the hyperlink (highlighted) that will post the encoded query string to our root proc if or when “Auto Login” is clicked.

Figure 1.3 modified page demonstrating the use of `encode` command;

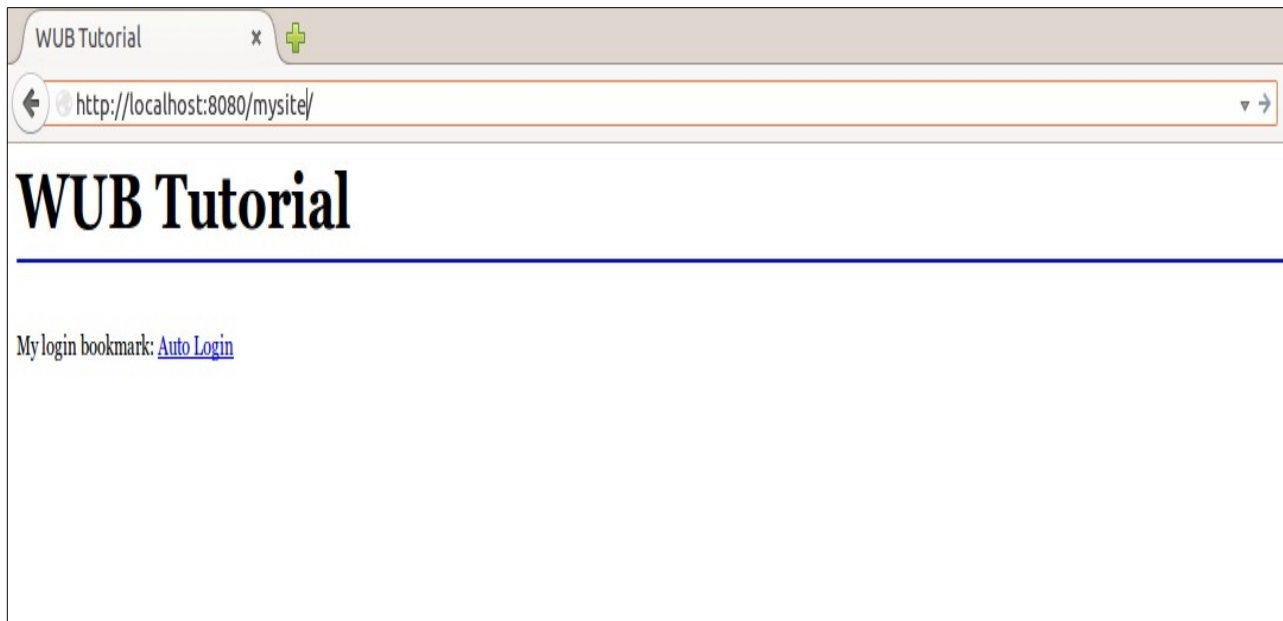
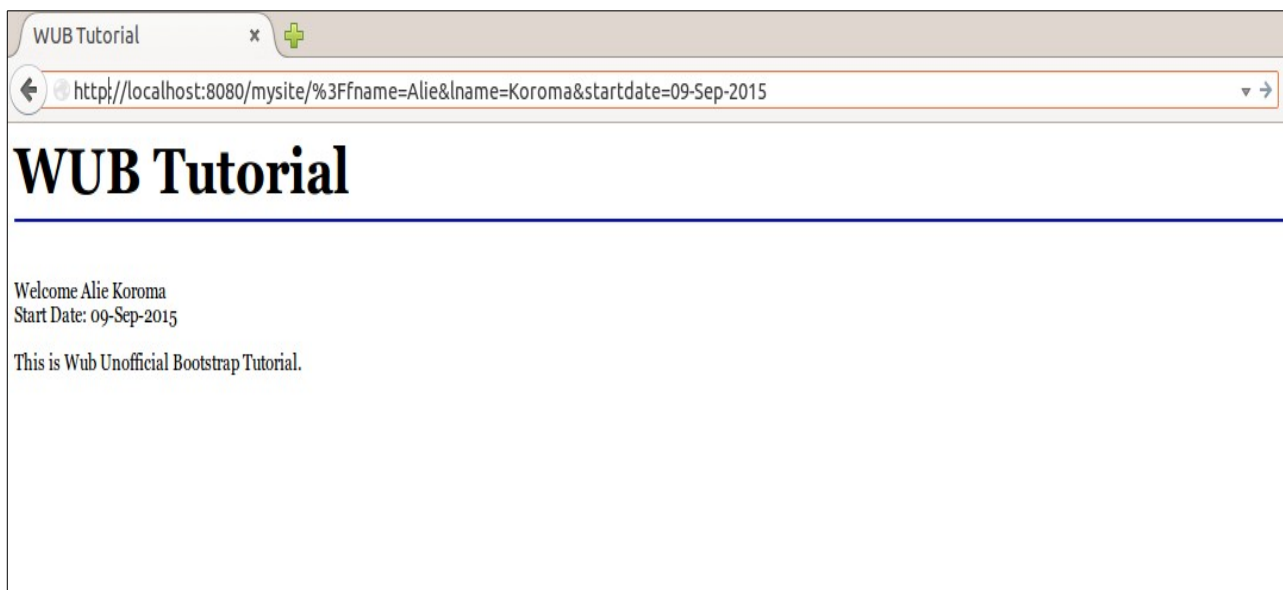


Figure 1.4 Auto Login was clicked and the values were used to login



Clicking the “**Auto Login**” in the previous page will post the query string we constructed using the `encode` command to Wub server. The server returned a page which included the values from the query string (*fname*, *lname* and *startdate*).

There are many more commands provided by the Query utility. There is even an `ncgi` compatible `nvlist` command available. Take a look at the source code in `/$install_dir/Utilities/Query.tcl` for more details.

Using Html

“the neatest html markup ever!”

Remember Wub is a web framework that provides tools (utilities and domains) that assist you in rapidly developing better web applications. HTML is a markup language, “*the neatest html markup ever!*” refers to Html’s tag generation capability.

We will rewrite the code that produced the pages above (Prac 1.4). From this point on, we will ditch the conventional <html> tags (can't wait...it saves me some typing) and transition into the neatest html markup ever!

Prac 1.4 rewrite Prac 1.3 in the neatest html markup ever!

```
1 namespace eval ::MySite {
2
3 proc / {r args} {
4
5     if {0} {
6         puts "root::MySite:RequestDict_r: -->\n $r"
7         set mypage {
8             <html>
9                 <head>
10                    <title>WUB Tutorial</title>
11                </head>
12                <body>
13                    <font size=14px><b>WUB Tutorial</b></font><br><hr color="light-blue"><br><br>
14                }
15            }
16
17            set mypage [<font> size 14px [<b> "WUB Tutorial" ]<br> [<br>] [<hr> color blue] [<br>]
18            [<br>]
19            if { [Query::exists [Query::parse $r] fname] } {
20
21                # extract the values from the query string or the request dict
22                set fName [Query::value [Query::parse $r] fname]
23                set lName [Query::value [Query::parse $r] lname]
24                set sDate [Query::value [Query::parse $r] startdate]
25
26                append mypage "Welcome $fName $lName<br>]Start Date: $sDate<br>] [<br>]"
27                append mypage "Wub Unofficial Bootstrap Tutorial. [<br>] [<br>]"
28
29            } else {
30
31                set encStr [Query::encode "?fname=Alie&lname=Koroma&startdate=[clock format
32                    [clock seconds] -format {%d-%b-%Y}]" ]
33                append mypage "My login bookmark: [<a> href $encStr {Auto Login}]"
34            }
35            #append mypage "</body></html>" #we don't need you anymore
36            dict append r -title "WUB Tutorial" ;# equivalent to line 9-11
37
38            return [Http Ok $r $mypage]
39        }
40
41        namespace export -clear *
42        namespace ensemble create -subcommands {}
43
44    }
```

Note that we didn't have to use `<html>`, `<body>` and `<head>` tags. I deliberately left the old code in there (commented out of course, lines 5-15 and 34) , so it's easy to make a comparison. In line 17 you can see that the commands are the same names as the html tags. There are no = signs for the html attributes (e.g `size=14px` becomes `size 14px`).

Also note in line 36 we set the page title by directly manipulating the request/response dictionary;

```
dict append r -title "WUB Tutorial"
```

Figure 1.5 Modified code using Wub Html commands instead of standard html

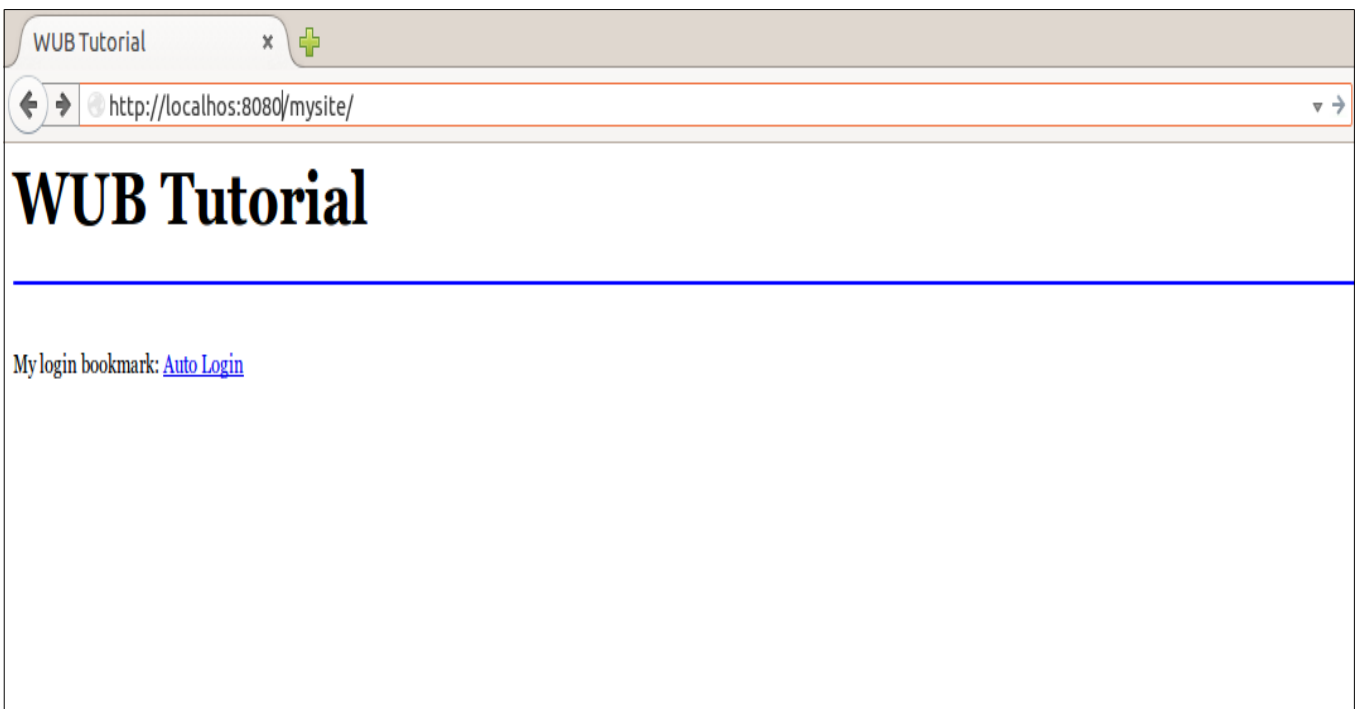
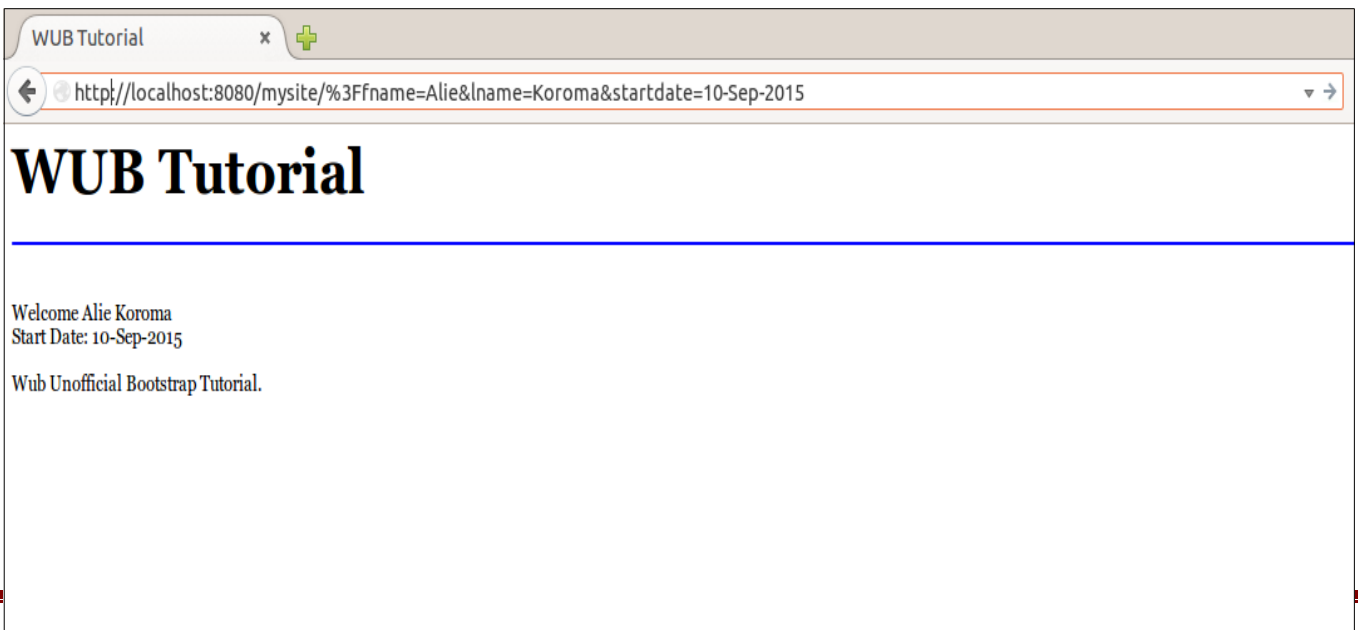


Figure 1.6 Modified page after clicking Auto Login



The syntax for Html tags is `<tagname> attr value attr value attr ...`

The command ` size 14px color green face arial` will yield the html output;

```
<font size='4px' color='green' face='arial'></font>
```

To return the value of the command you put them in braces;

```
set mypage [<font> size 14px color green face arial]
```

“What about tables, menu and style?”

Tables and menus sounds like we talking about restaurants. Table is a valid html tag frequently used for organizing or presenting content on a page. Menus can also be created with a simple command. We will demonstrate a few more useful commands like `<table>`, `<menulist>`, `<stylesheet>`, `<div>`, `<links>` etc. in the next Prac section.

Prac 1.5 more demonstration of the Html utility

```
namespace eval ::MySite {

    proc / {r args} {
        puts "root::MySite:RequestDict_r: -->\n $r"

        # Build the page header taking into account the different query strings passed
        # from menulist
        if { [Query::exists [Query::parse $r] fname] } {

            # extract the values from the query string or the request dict
            set fName [Query::value [Query::parse $r] fname]
            set lName [Query::value [Query::parse $r] lname]
            set sDate [Query::value [Query::parse $r] startdate]

            append mypage "Welcome: $fName $lName<br>Today's Date: $sDate<br>
            <br>"
            set mypage [<div> [<h2> {WUB Tutorial}]$mypage]

        } else {

            set mypage [<div> "[<h1> {WUB Tutorial}]Wub Unofficial Bootstrap
            Tutorial<br>pure-Tcl HTTP 1.1 Server"

        ]

        append mypage [<br>][<br>][<hr> color blue] [<br>]
        append mypage [<b> {Menulist:}] ;#returns string Menulist: in bold
        set encStr [Query::encode "?fname=Alie&lname=Koroma&startdate=[clock format
        [clock seconds] -format {%d-%b-%Y}"]"]
        append mypage [Html::menulist "Home / AutoLogin $encStr LogOut [Query::encode
        logout]" ] [<br>]
        append mypage [<div> width 100% height 400px border 1 [<p> [<h2> {Wub image;}]]]
        append mypage [<div> [<img> src /images/wub_tcl_on_web.jpg width 90px height
        120px]] [<br>][<br>]

        append mypage [<div> width 100% height 400px border 1 [<p> [<h2> {a sample
        table;}]]]
        append mypage [<tr> [<td> [<b> #]] [<td> [<b> Field]] [<td> [<b> Value]] ]
        append mypage [<tr> [<td> [<a href [Query::encode ?#] 1]] [<td> "Tcl Version:"
        [<td> "Tcl8.6/[info patchlevel]" ] ]
        append mypage [<tr> [<td> [<a href [Query::encode ?#] 2]] [<td> "Script:" [<td>
        [info script]] ] ]
        append mypage [<tr> [<td> [<a href [Query::encode ?#] 3]] [<td> "Procedures:"
        [<td> [info proc]] ] ]
        append mypage [<tr> [<td> colspan 3] ] ]
        append mypage [<tr> [<td> colspan 3 {Dict Dump:}] ] ]
        append mypage [<tr> [<td> colspan 3 $r] ] ]

        set mypage [<div> [<table> border 1 width 100% $mypage]]
        append mypage [<div> height 80px "[<br>][<br>][<hr> color blue][<br>]TclWeb [<i>
        {powered by Wub}]]]"
        set mypage [<div> $mypage];#add page wrapper

        # ditch the tags and make the code look more tcl
        dict append r -title "WUB Tutorial" ;#add page title

        return [Http Ok $r $mypage]

    }

}
```

```

proc /logout {r args} {
    set logout [<font> size 14px [<b> "WUB Tutorial"]][<br>][<br>][<hr> color
blue][<br>]
    append logout [<p> [<b> "Logged Out"]][<br>][<br>][<br>][<a href
[Query::encode /] {Start Over}]]
    dict append r -title "WUB Tutorial"

    return [Http Ok $r $logout]
}

namespace export -clear *
namespace ensemble create -subcommands {}
}

```

Menu lists are easy to create with the `Html` utility, the following excerpt from `Prac 1.5` shows how this was done;

```

set encStr [Query::encode "?fname=Alie&lname=Koroma&startdate=[clock format [clock seconds]
-format {%d-%b-%Y}]]"]

append mypage [Html::menulist "Home / AutoLogin $encStr Logout [Query::encode logout]"]

```

The first line builds/encodes a query string where `fname=Alie` and `lname=Koroma` and `startdate` is today's date.

The query string does not have to be encoded in some cases but I like to do it anyway. This eliminates the confusion of when to and when not to. The following line will also work;

```

set encStr "?fname=Alie&lname=Koroma&startdate=[clock format [clock seconds] -format {%d-%b-%Y}]]"

```

The second line of code starting with `append` above builds a bulleted menulist as follows;

```

Home (Home /): when clicked will call proc /
AutoLogin: when clicked will post a query string stored in $encStr (?fname=A...)
Logout (Logout [Query::encode logout]): when clicked will call proc /logout

```

If your query string is to call a `proc` like `/logout` for example, it must be encoded. However, you can get away with not encoding the forward slash that calls `proc /`. If we had "Home {}" it will still call `proc /`. On the other hand, if you fail to encode `/logout`, you will get the following error;

```

http://localhost:8080/logout Not Found

```

These are the intricacies we trying to avoid thus, the reason for encoding all query strings. The `image`, `table` and `div` are straight-forward, they are used in exactly the same way you write `html`, the only difference is the attribute do not have equals sign.

Figure 1.7 modified page displaying the use of menulist, table, image and div elements

WUB Tutorial

localhost:8080/mysite/


WUB Tutorial

Wub Unofficial Bootstrap Tutorial
pure-Tcl HTTP 1.1 Server

Menulist:

- [Home](#)
- [AutoLogin](#)
- [LogOut](#)

Wub image;



a sample table;

#	Field	Value
1	Tcl Version:	Tcl8.6/8.6.1
3	Procedures:	/logout /

Dict Dump:

```
-host localhost -port 8080 -htpd {::Httpd new} -id ::oo::Obj51 -server {127.0.0.1 127.0.0.1 8080} -scheme http -sock sock29dd5a0 -cid ::oo::Obj59 -ipaddr 127.0.0.1 -rport 51692 -received_sec
-server_id {Wub 6.0} -pipeline 4 -send ::Httpd::coros::obj59 -transaction 4 -time {connected 1442630144599376} -header {GET /mysite/ HTTP/1.1} -method GET host localhost:8080 -clienthe
-user-agent accept accept-language accept-encoding connection user-agent {Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0} accept {text/html,application/xh
/xml;q=0.9,*/*;q=0.8} accept-language {en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3} accept-encoding {gzip, deflate} connection keep-alive -version 1.1 -ua {ua {Mozilla/5.0 (X11; Ubuntu; Lin
Gecko/20100101 Firefox/31.0) id FF version {} mozilla_version 5.0 extensions {} platform X11 security Ubuntu subplatform {Linux x86_64} language rv:31.0 product {Gecko 20100101 Firefox
unknown -normalized 1 -path /mysite/ -url http://localhost:8080/mysite/ -uri http://localhost:8080/mysite/ -forwards {} -encoding binary -received 1442630176681177 -cookies {} -Query {}
-suffix {} -extension {} content-type x-text/html-fragment -dynamic 1 -extra {} -fprefix / -cprefix /
```

TclWeb powered by Wub

“the page looks plain, what about some style?”

Of course Wub supports cascading style sheets (CSS). The implementation especially with Direct Domains is pretty cool as it embeds what will be your style-sheet file (with the .css extension) into a single proc. Below we demonstrate how the previous lifeless, dull and boring looking page could be brought to life with a little colour and style using CSS.

You will notice all the html formatting (e.g font-size 14px etc...) have disappeared from our code and now resides in a proc call ***/mysitecss***. This proc is our css file per se. For this one, We'll show the page before the code as seen below;

Figure 1.8 modified page using cascading style sheet

WUB Tutorial

Wub Unofficial Bootstrap Tutorial
pure-Tcl HTTP 1.1 Server

WUB
Tcl on WEB

Where is Wub?

Wub is hosted here: <http://code.google.com/p/wub/>
Wub has a discussion group here: <http://groups.google.com/group/wub-discussion>
Wub code can be fetched from svn with svn checkout <http://wub.googlecode.com/svn/trunk/wub>
Occasional releases can be found here: 1 Issues with Wub can be reported here: 2 Wub tutorial presented at 8th European Tcl/Tk Users Meeting, available from <http://code.google.com/p/wubwiki/downloads/list> Follow that link and download either eurotel2009-examples.zip or eurotel2009-wub-tcllib-examples.zip, those two archives contain a PDF file named Lets_Wub.pdf, that is the tutorial.

- Home
- AutoLogin
- LogOut

Installation Details:

#	Field	Value
1	Tcl Version:	Tcl8.6/8.6.1 Required/Installed
2	Script:	./Wub.tcl
3	Namespace Procedure(s):	/logout /mysitcess /

Dict Dump:

```
-host localhost -port 8080 -httpd {::Httpd new} -id ::oo::Obj51 -server {127.0.0.1 127.0.0.1 8080} -scheme http -sock sock16beedo -cid ::oo::Obj57 -ipaddr 127.0.0.1 -rport 60794 -received_seconds 1442730050 -server_id {Wub 6.0} -pipeline 3 -send ::Httpd::coros::obj57 -transaction 2 -time {connected 1442730050643783} -header {GET /mysite/ HTTP/1.1} -method GET host localhost:8080 -clientheaders {host user-agent accept accept-language accept-encoding referer connection} user-agent {Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0} accept {text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8} accept-language {en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3} accept-encoding {gzip, deflate} referer http://localhost:8080/mysite/%3Ffname=Alie&lname=Koroma&startdate=20-Sep-2015 connection keep-alive -version 1.1 -ua {ua {Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0} id FF version {} mozilla_version 5.0 extensions {} platform X11 security Ubuntu subplatform {Linux x86_64} language rv:31.0 product {Gecko 20100101 Firefox 31.0}} -ua_class unknown -normalized 1 -path /mysite/ -url http://localhost:8080/mysite/ -uri http://localhost:8080/mysite/ -forwards {} -encoding binary -received 1442730050648691 -cookies {} -Query {} -prefix /mysite/ -suffix {}
```

The menulist items are activated as your mouse pointer hovers over them. Let's take a look at the code behind the above page and the CSS implementation in particular.

Prac 1.6 applying some style to the page without changing the content much

```
namespace eval ::MySite {

    proc / {r args} {

        # Build the page header taking the value may be passed from the menulist
        if { [Query::exists [Query::parse $r] fname] } {

            # extract the values from the query string or the request dict
            set fName [Query::value [Query::parse $r] fname]
            set lName [Query::value [Query::parse $r] lname]
            set sDate [Query::value [Query::parse $r] startdate]

            append mypage "<b> Welcome:] $fName $lName[<br>]Today's Date: $sDate[<br>][<br>]"
            set mypage [<div> id divtop [<h2> class wubhead {WUB Tutorial}]$mypage[<br>]]

        } else {

            set mypage [<div> id divtop "[<h2> class wubhead {WUB Tutorial}][<b> class bwub
            W]ub\ [<b> class bwub U]nofficial\ [<b> class bwub B]ootstrap Tutorial[<br>]pure-
            Tcl HTTP 1.1 Server[<br>][<br>][<hr>]"

        }

        set encStr [Query::encode "?fname=Alie&lname=Koroma&startdate=[clock format [clock
        seconds] -format {%d-%b-%Y}]" ]

        set mymenu [<img> class img src /images/wub_tcl_on_web.jpg width 90px height 120px]
        append mymenu [Html::menulist "Home / AutoLogin $encStr LogOut [Query::encode logout]" ]
        set mymenu [<div> id divmenu [<p> class menu $mymenu]]

        if { [Query::exists [Query::parse $r] fname] } {

            set wubintro [<div> id divhome "
            [<h1> {What is Wub: Introduction}]
            Wub is a web-server written in pure-Tcl. It runs the Wiki you're now
            reading.
            It absolutely requires Tcl 8.6 and tracks the HEAD closely. It should help
            in creating highly-dynamic (and portable) web applications.
            It is the successor in spirit of Tclhttpd, aiming to preserve the best of it
            while using the bleeding-edge latest Tcl facilities to simplify and extend.
            Wub is essentially an interpreter for HTTP 1.1 requests, and it's very
            appropriate to use Tcl for that kind of thing. Here's Wub's Read-Eval-Print
            loop ... couldn't be simpler.
            [<br>][<br>][<img> src /images/arch.png class imgarch]
            "]"
            append mypage [<div> id divbody $mymenu$wubintro]

        } else {

            set mytable [<tr> class tabhead [<td> [<b> #]] [<td> [<b> Field]] [<td> [<b>
            Value]] ]
            append mytable [<tr> [<td> [<a> href [Query::encode ?#] 1]] [<td> "Tcl Version:"
            [<td> "Tcl8.6/[info patchlevel] Required/Installed" ] ]
            append mytable [<tr> [<td> [<a> href [Query::encode ?#] 2]] [<td> "Script:" [<td>
            [info script]] ] ]
            append mytable [<tr> [<td> [<a> href [Query::encode ?#] 3]] [<td> "Namespace
            Procedure(s):" [<td> [info proc]] ] ]
            append mytable [<tr> [<td> colspan 3 ] ]
            append mytable [<tr> [<td> colspan 3 {Dict Dump:}] ] ]
            append mytable [<tr> [<td> colspan 3 $r] ] ]

            set whereiswub [<div> id divhome "
            [<h1> {Where is Wub?}]
            Wub is hosted here [<a> href http://code.google.com/p/wub/
            {http://code.google.com/p/wub/}] [<br>]
            Wub has a discussion group here [<a> href http://groups.google.com/group/wub-
            discussion {http://groups.google.com/group/wub-discussion}] [<br>]
            Wub code can be fetched from svn with svn checkout
            http://wub.googlecode.com/svn/trunk/ wub[<br>]
            Occasional releases can be found here: 1
            Issues with Wub can be reported here: 2
            Wub tutorial presented at 8th European Tcl/Tk Users Meeting, available from [<a>
```

```

        href http://code.google.com/p/wubwikit/downloads/list
        {http://code.google.com/p/wubwikit/downloads/list}}
        Follow that link and download either eurotcl2009-examples.zip or eurotcl2009-wub-
            tcllib-examples.zip,
        those two archives contain a PDF file named Lets_Wub.pdf, that is the tutorial.
        [<br>]<br>
        [<h2> {Installation Details:}]
        [<table> class mytable $mytable]
        "]"
        append mypage [<div> id divbody $ymenu$whereiswub]
    }

    set myfoot "[<br>TclWeb [<i> {powered by Wub}][<br>][<a> href http://wiki.tcl.tk/15781
        {Wub wiki page}]"
    set myfoot [<hr>]$myfoot
    append mypage [<div> id divfoot $myfoot]

    set mypage [<div> $mypage] ;#add page wrapper

    dict append r -title "WUB Tutorial" ;# add a page title

    set r [Html style $r mysitecss]
    return [Http Ok $r $mypage]
}

proc /logout {r args} {

    set logout [<h2> class wubhead {WUB Tutorial}][<br>]
    append logout [<p> [<b> "Logged Out..."]<br>][<br>][<a> href [Query::encode /] {Start
        Over}] ]
    set logout [<div> id divtop $logout<hr>]

    dict append r -title "WUB Tutorial"

    set r [Html::style $r mysitecss]
    return [Http Ok $r $logout]
}

proc /mysitecss {r args} {

    set css {
        a { font-size:12px; font-weight:bold; text-decoration:none; text-underline:none; }
        div#divbody { width:100%; height:auto; padding-right:50px; }
        div#divfoot { width:100%; height:100px; background-color:#43e8d3; clear:both; text-
            align:center; border-radius:5px; padding:20px; margin:5px; }
        div#divhome { width:70%; float:left; padding:20px; margin:5px; }
        div#divtop { padding:15px; width:100%; height:auto; background-color:#43e8d3;
            border-radius:5px; }
        div#divmenu { width:18%; height:auto; float:left; }
        div#divtable { width:60%; height:auto; float:left; padding:20px; }
        .bwub { color:black; font-weight:bold; font-size:16px; }
        .tabhead { background-color:#43e8d3; font-weight:bold; }
        .img { width:auto; height:120px; padding:20px; margin:2px; }
        .imgarch { width:600px; height:600px; clear:both; }
        .menu { font-size:12px; }
        .mytable { width:100%; height:auto; text-align:center; border:1px solid #43e8d3; }
        .wubhead { font-size:18px; font-weight:bold; color:#fff; }
        hr { color:#43e8d3; width:100%; height:5px; box-shadow:3px 3px 3px #fff;
            border:0px; border-radius:3px; }
        td { border:1px solid #43e8d3; border-radius:5px; }
        a:hover { border:0px; background-color:#43e8d3; cursor:grab; font-size:12px; font-
            weight:bold;
            color:#fff; border-radius:5px; box-shadow: 3px 3px 3px grey; padding:2px;
        }
        tr:hover { background-color:#43e8d3; font-weight:bold; }
    }

    return [Http Ok $r $css text/css]
}

namespace export -clear *
namespace ensemble create -subcommands {}
}

```

Of the whole of Prac 1.6, we are only interested in one proc and one line of statement to explain the implementation of CSS in Wub. Within your proc, define all your CSS attributes and return it in a Http Ok response. Before sending the page response, style it with the `Html style` command to apply the CSS to your page.

Defining the CSS proc is easy. The following `proc /mysitecss` returns all the style sheet definition in a variable called `mycss` as a Http Ok response.

```
proc /mysitecss {r args} {  
  
    set mycss {  
        a { font-size:12px; font-weight:bold;  
            text-decoration:none;  
            text-underline:none;  
        }  
        . . . . .  
    }  
    return [Http Ok $r $mycss text/css]  
}
```

You will then apply this style to your page before sending your response as follows;

```
set r [Html style $r mysitecss]  
    or  
set r [Html::style $r mysitecss]
```

This will set the `-style` element of the request/response dict to;

`-style {mysitecss {}}`

The traditional method is also supported with the `<stylesheet>` command. If your CSS is defined in a file called `mysitecss.css` in the root directory for example. You can reference it within your code as follows;

```
<stylesheet> ./mysitecss.css
```

This will return the html tag;

```
<link rel='StyleSheet' type='text/css' media='screen' href='./mysitecss.css'>
```

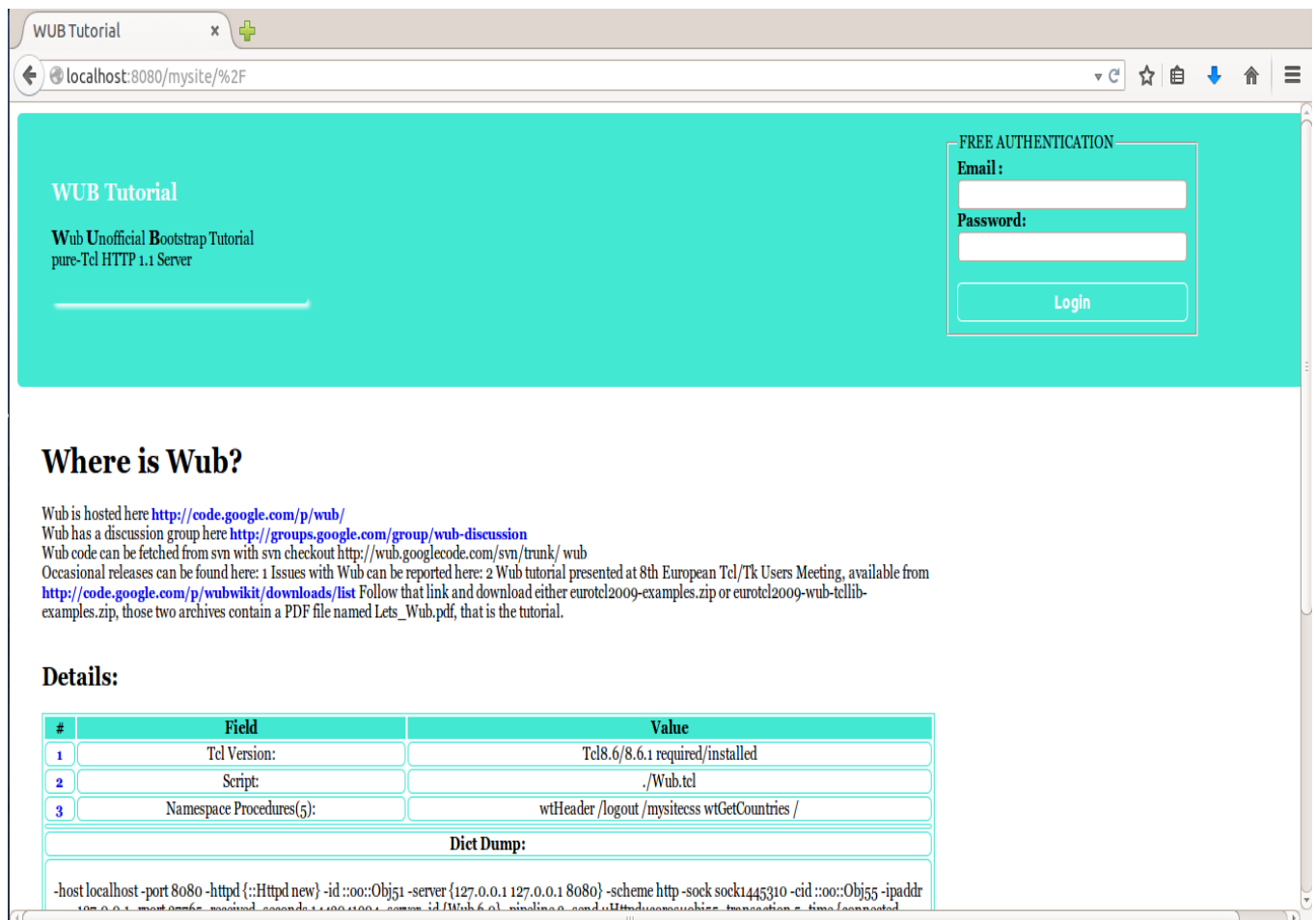
There are many more commands provided by the Html utility like `If`, `Foreach`, `Switch`, `While`, `dict2json`, `dict2table` to name a few.

Using Form

“build them...whatever the shape or form”

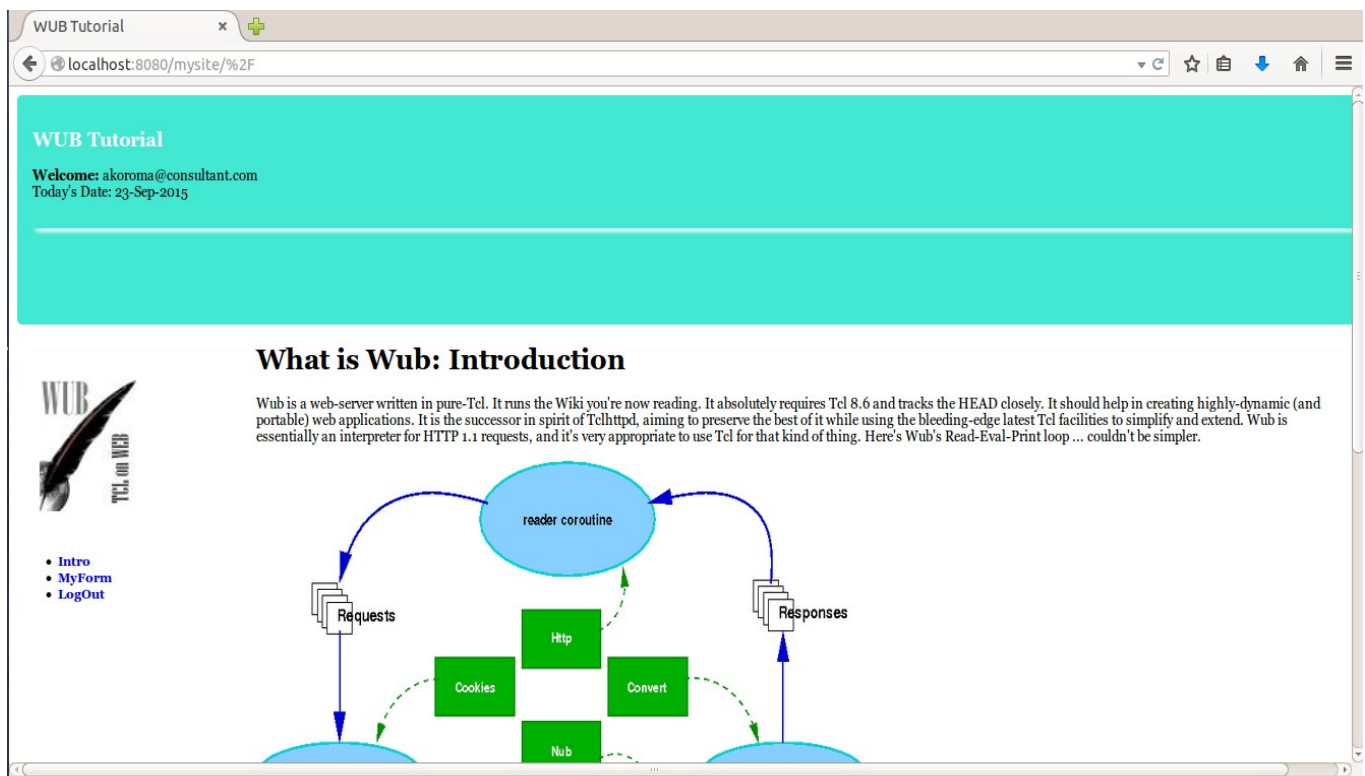
Like most other things in Wub, Forms are pretty easy to build using the **Form** utility. It provides commands like `<checkbox>`, `<password>`, `<textarea>`, `<hidden>` and many more. For now we will replace the AutoLogin with an actual user input *username* and *password* from a form (simulating user login).

Figure 1.9 modified page using a form for user authentication (login)



If you click login after any value was inserted in the Email field, you are logged in to the page with a menulist on the left hand side and the “*What is Wub*” content on the right. The user session is maintained with a single variable called *isLoggedIn* until the **LogOut** menu item is clicked.

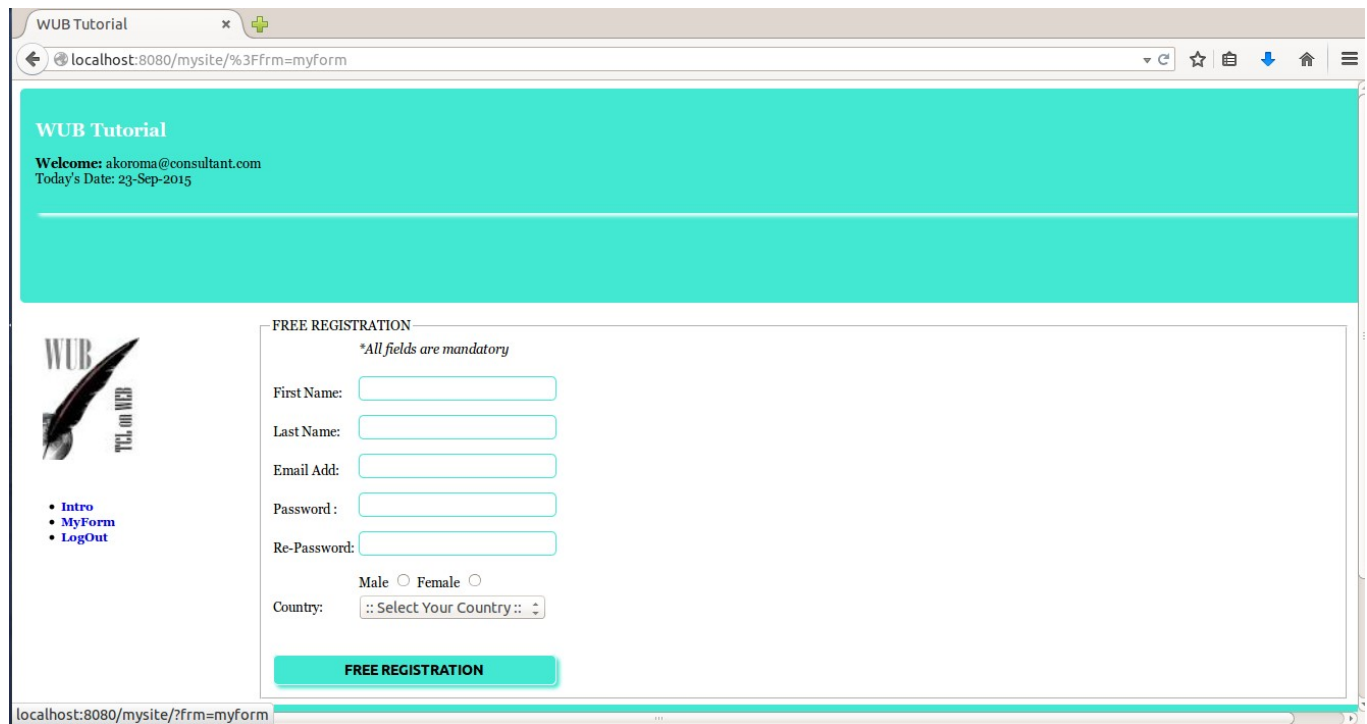
Figure 2.0 the intro page after login



If you click on the Intro link, it will reload the same page.

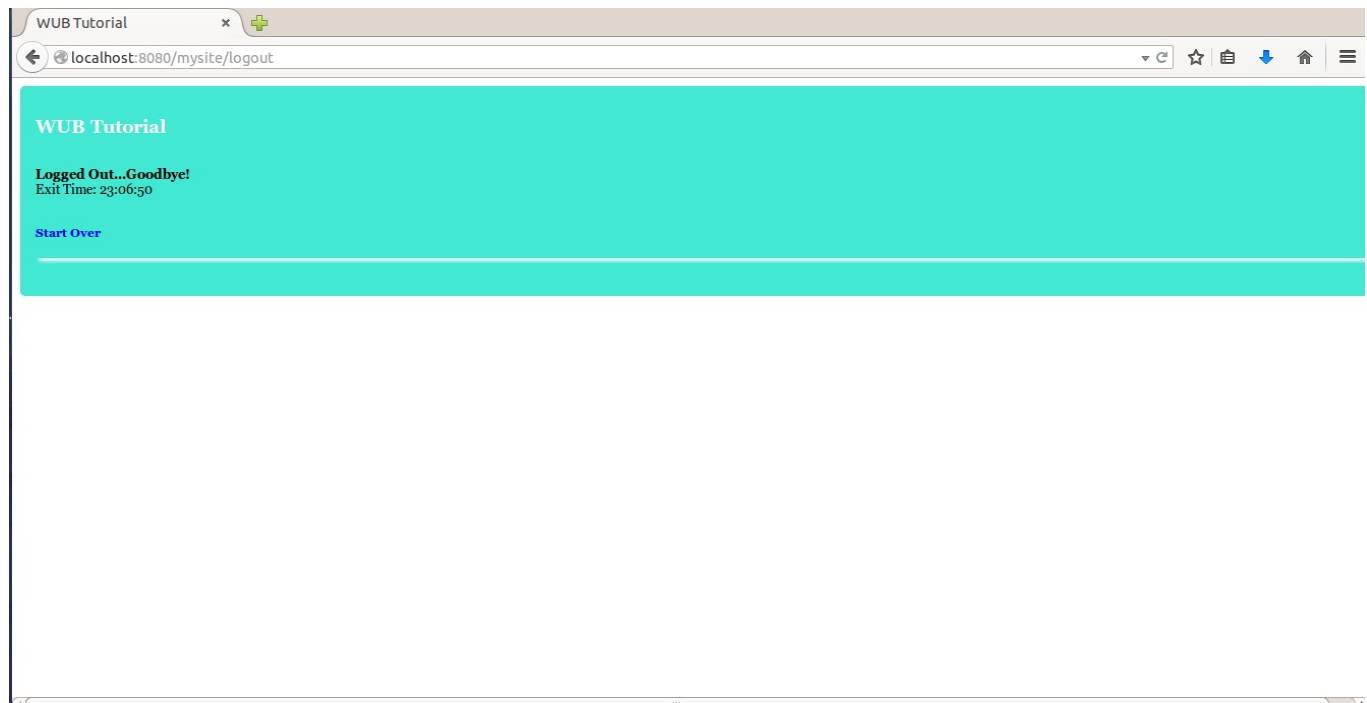
The **MyForm** menu item displays a form with different field types like text, radioset (a set of radio buttons), password, select and a submit button.

Figure 2.1 the MyForm page after login



The **LogOut** menu item will terminate your session (set isLoggedIn=0) and log you out to a page as seen below;

Figure 2.2 the LogOut page after login



We will take our focus back to the FREE AUTHENTICATION heading with the login button in **Fig 1.9**. It's a form that uses the HTTP post method to submit its values. Below is the code for the pages illustrated above, we will visit the form construct section immediately after.

Prac 2.1 using form to add user authentication to the main page

```
namespace eval ::MySite {

    variable isLoggedInIn 0          ;#flag if the user is logged in or not
    variable remMyHeader {}         ;#remember the header for use in subsequent user activitiesd

    proc / {r args} {

        variable isLoggedInIn
        variable remMyHeader

        # Build the page header taking the value may be passed from the menulist
        if { [Query::exists [Query::parse $r] username] && ![string equal [Query::value [Query::parse $r] username] {}] } {
            set mypage [wtHeader [Query::value [Query::parse $r] username] [Query::value [Query::parse $r] usrpasswd] ]
            set remMyHeader $mypage
            set isLoggedInIn 1
        } elseif { $isLoggedInIn == 1 } {
            set mypage $remMyHeader          ;#set the header to the authenticated one
        } else {
            set mypage [wtHeader]
        }

        if { $isLoggedInIn == 1 } {          ;#the user is still logged in
            if { [Query::exists [Query::parse $r] frm] } {

                set wubform [<tr> [<td>]<td colspan 2 <i>{*this is a demo form, all field are mandatory}]]
                append wubform [<tr> [<td colspan 2 [<hidden> hideme id hideme]]
                append wubform [<tr> [<td> "First Name:"<td> [<text> fname id fname class frmfld title {First Name}]]
                append wubform [<tr> [<td> "Last Name:"<td> [<text> lname id lname class frmfld title {Last Name}]]
                append wubform [<tr> [<td> "Email Add.:"<td> [<text> email id email class frmfld title {Email Address}]]
                append wubform [<tr> [<td> "Password ::"<td> [<password> passwd id passwd class frmfld title {Password}]]
                append wubform [<tr> [<td> "Re-Password:"<td> [<password> repeat id repeat class frmfld title {Retype Password}]]
                append wubform [<tr> [<td> "Radioset : ":"<td> [<radioset> mysex id mysex {Male male Female female}]]
                append wubform [<tr> [<td> "Checkset : ":"<td> [<checkboxset> mysurvey id mysurvey {Newsletter lnews Whitepaper wpaper}]]
                append wubform [<tr> [<td> "Country:"<td> [wtGetCountries]]
                append wubform [<tr> [<td colspan 2 [<br>]<br>]<submit> subReg id subReg class frmbutton {FREE REGISTRATION}]]
                set wubform [<table> $wubform]

                # build the form and insert the fields contained in variable 'wubform'
                set wubform [<br>]<form> frmlogin id frmlogin name frmlogin action [Query::encode "?/signup"] method POST {
                    [<fieldset> { [<legend> { FREE REGISTRATION }} $wubform ]}
                }

                set wubform [<div> $wubform]

            } elseif { [Query::exists [Query::parse $r] subReg] } {
                # respond to the free registration form post

                set myReg [<p> "
                    {If { [string equal [Query::value [Query::parse $r] fname] {}] } {
                        "Guest"
                    } else { "[Query::value [Query::parse $r] fname]" }},
                    [<br>]<b> class bfont {thanks for registering with us.}}
                "
            ]

            } else {

                set wubintro [<div> "
                    [<h1> {What is Wub: Introduction}}
                    Wub is a web-server written in pure-Tcl. It runs the Wiki you're now reading.
                    It absolutely requires Tcl 8.6 and tracks the HEAD closely. It should help in creating
                    highly-dynamic (and portable) web applications.
                    It is the successor in spirit of Tclhttpd, aiming to preserve the best of it while using the
                    bleeding-edge latest Tcl facilities to simplify and extend.
                    Wub is essentially an interpreter for HTTP 1.1 requests, and it's very appropriate to use
                    Tcl for that kind of thing. Here's Wub's Read-Eval-Print loop ... couldn't be simpler.
                    [<br>]<br>]<img> src /images/arch.png class imgarch
                "
            ]

            set mymenu [<img> class img src /images/wub_tcl_on_web.jpg width 90px height 120px]
            append mymenu [Html::menulist "Intro / {MyForm} [Query::encode ?frm=myform] Logout [Query::encode logout]" ]
            set mymenu [<div> id divmenu [<p> class menu $mymenu]

            if { [Query::exists [Query::parse $r] frm] } {
                append mypage [<div> id divbody $mymenu$wubform]
            } elseif { [Query::exists [Query::parse $r] subReg] } {
                append mypage [<div> id divbody $mymenu$myReg]
            } else {
                append mypage [<div> id divbody $mymenu$wubintro]
            }

        } else {

            # create a table with field values and store it in a variable mytable for use in the following section
            set mytable [<tr> class tabhead [<td> [<b> #]]<td> [<b> Field]]<td> [<b> Value]] ]
            append mytable [<tr> [<td> class tabdetail [<a> href [Query::encode ?#] 1]]<td> class tabdetail "Tcl Version:"<td> class tabdetail "Tcl8.6/[info patchlevel] required/installed" ]
        }
    }
}
```

```

append mytable [<tr> [<td> class tabdetail [<a> href [Query::encode ?#] 2]] [<td> class tabdetail "Script:"
[<td> class tabdetail [info script]] ]
append mytable [<tr> [<td> class tabdetail [<a> href [Query::encode ?#] 3]] [<td> class tabdetail "Namespace
Procedures\([length [info proc]]\):" ] [<td> class tabdetail [info proc]] ]
append mytable [<tr> [<td> class tabdetail colspan 3 ] ]
append mytable [<tr> [<td> class tabdetail colspan 3 [<b> {Dict Dump:}]] ]
append mytable [<tr> [<td> class tabdetail colspan 3 [<br>] $r [<br>] [<br>] ] ]

set whereiswub [<div> id divhome "
[<h1> {Where is Wub?}]
Wub is hosted here [<a> href http://code.google.com/p/wub/ {http://code.google.com/p/wub/}] [<br>]
Wub has a discussion group here [<a> href http://groups.google.com/group/wub-discussion
{http://groups.google.com/group/wub-discussion}] [<br>]
Wub code can be fetched from svn with svn checkout http://wub.googlecode.com/svn/trunk/ wub [<br>]
Occasional releases can be found here: 1
Issues with Wub can be reported here: 2
Wub tutorial presented at 8th European Tcl/Tk Users Meeting, available from [<a> href
http://code.google.com/p/wubwikit/downloads/list {http://code.google.com/p/wubwikit/downloads/list}]
Follow that link and download either eurotcl2009-examples.zip or eurotcl2009-wub-tcllib-examples.zip,
those two archives contain a PDF file named Lets_Wub.pdf, that is the tutorial.
[<br>] [<br>]
[<h2> {Details:}]
[<table> class mytable $mytable]
"]
append mypage [<div> id divbody $whereiswub
]

set myfoot "[<br>]TclWeb [<i> {powered by Wub}] [<br>] [<a> href http://wiki.tcl.tk/15781 {Wub wiki page}]"
set myfoot [<hr>] $myfoot
append mypage [<div> id divfoot $myfoot]

set mypage [<div> $mypage] ;#add page wrapper

dict append r -title "WUB Tutorial" ;# add a page title

set r [Html style $r mysitecss] ;# add style called mysitecss to response r
return [Http Ok $r $mypage] ;# send a HTTP Ok response
}

proc wtHeader { {username 0} {passwd 0} } {
# Build the page header taking the value may be passed from the menulist
if { $username != 0 } {
set pheader [<h2> class wubhead {WUB Tutorial}]
append pheader "[<b> Welcome:] $username [<br>] Today's Date: [clock format [clock seconds] -format {%d-%b-%Y}]
[<br>] [<br>] [<hr>]"
return [<div> id divtop $pheader]
} else {
set ptitle "
[<h2> class wubhead {WUB Tutorial}] [<b> class bwub W]ub [ [<b> class bwub U]nofficial\ [ [<b> class bwub
B]ootstrap Tutorial [<br>]
pure-Tcl HTTP 1.1 Server [<br>] [<br>] [<hr>]"
# build the form fields in a table for formatting
#set p_html [<b> {USER LOGIN}] [<br>] [<font> class slogan {Free authentication}] [<br>] [<br>]
set p_html [<b> {Email : } ] [<text> username id usrname class fld title "email address" tabindex 1] [<br>]
append p_html [<b> {Password: } ] [<password> usrpasswd id usrpas class fld title "password" tabindex 2] [<br>]
[<br>]
append p_html [<submit> smtlogin id smtlogin value Login tabindex 3 class lbutton Login]
set pheader [<form> frmlogin action [Query::encode /] method get class login {
[<fieldset> class hdr { [<legend> { FREE AUTHENTICATION }}] $p_html }}
]]
return [<div> id divtop [<div> id divtitle $ptitle] [<div> id divtopright $pheader] ]
}
}

proc /logout {r args} {
set logout [<h2> class wubhead {WUB Tutorial}] [<br>]
append logout "[<b> {Logged Out...Goodbye!}] [<br>] Exit Time: [clock format [clock seconds] -format {%H:%M:%S}] [<br>]
[<br>]"
append logout [<p> [<a> href [Query::encode /] {Start Over}] ]
set logout [<div> id divtop $logout [<hr>]]
variable isLoggedIn 0
dict append r -title "WUB Tutorial"
set r [Html::style $r mysitecss]
return [Http Ok $r $logout]
}

proc wtGetCountries { } {
set allCountry [<option> value neselect {:: Select Your Country ::}]
foreach {cName} [list Algeria Benin Columbia Denmark ZA] {
append allCountry [<option> value $cName $cName]
}
return [<select> country class fld $allCountry]
}
}

```

```

proc /mysitecss {r args} {

    set css {
        a { font-size:12px; font-weight:bold; text-decoration:none; text-underline:none; }
        div#divbody { width:100%; height:auto; padding-right:50px; }
        div#divfoot { width:100%; height:100px; background-color:#43e8d3; clear:both; text-align:center; border-
            radius:5px; padding:20px; margin:5px; }
        div#divhome { width:70%; clear:both; padding:20px; margin:5px; }
        div#divtop { padding:15px; width:100%; height:180px; background-color:#43e8d3; border-radius:5px; }
        div#divtopright { padding-right:135px; margin-right:40px; width:15%; height:auto; float:right; background-
            color:#43e8d3; }
        div#divmenu { width:18%; height:auto; float:left; }
        div#divtable { width:60%; height:auto; float:left; padding:20px; }
        div#divtitle { width:20%; height:auto; float:left; padding:20px; }
        .bfont { font-size:35px; font-weight:bold; color:#43e8d3; }
        .bwub { color:black; font-weight:bold; font-size:16px; }
        .frmfld { border:1px solid #43e8d3; border-radius:5px; text-align:left; margin-bottom:10px; height:20px;
            width:190px; }
        .thead { background-color:#43e8d3; font-weight:bold; }
        .img { width:auto; height:120px; padding:20px; margin:2px; }
        .imgarch { width:600px; height:600px; clear:both; }
        .frmbutton {
            border:1px solid #fff; border-radius:5px;
            width:100%; height:30px; font-size:14px; font-weight:bold; box-shadow:3px 3px 3px #43e8d3;
            right:20px; bottom:20px; background-color:#43e8d3; color:#000000; cursor:grab;
        }
        .lbutton {
            border:1px solid #fff; border-radius:5px;
            width:100%; height:30px; font-size:14px; font-weight:bold;
            right:20px; bottom:20px; background-color:#43e8d3; color:#fff; cursor:grab;
        }
        .menu { font-size:12px; }
        .mytable { width:100%; height:auto; text-align:center; border:1px solid #43e8d3; }
        .tabdetail { border:1px solid #43e8d3; border-radius:5px; }
        .wubhead { font-size:18px; font-weight:bold; color:#fff; }
        hr { color:#43e8d3; width:100%; height:5px; box-shadow:3px 3px 3px #fff; border:0px; border-radius:3px; }
        a:hover { border:0px; background-color:#43e8d3; cursor:grab; font-size:12px; font-weight:bold;
            color:#fff; border-radius:5px; box-shadow: 3px 3px 3px grey; padding:4px;
        }
        #tr:hover { background-color:#43e8d3; font-weight:bold; }
    }
    return [Http Ok $r $css text/css]
}

namespace export -clear *
namespace ensemble create -subcommands {}
}

```

The code below constructed the login form

```
set pheader [<form> frmlogin action [Query::encode /] method post {
    [<fieldset> { $p_html [<legend> { FREE AUTHENTICATION }} ]}]
}]
```

The command takes the form `<form> name {attrs} content`. In the above code;

<form>: is the command to construct a form

frmlogin: the name assigned to this form

action [Query::encode /]: when the form is submitted, the `proc /` is called and the query string values passed to it as arguments.

method post: HTTP method to use POST/GET when submitting the form. The difference between post and get is POST does not display the query string being sent and GET displays it in the address bar.

`<form>` command encapsulates a set of fields which are defined by the `<fieldset>` command. `<fieldset>` command also follow the same convention as `<form>` command with name, attributes and then contents. However, we already have the structure of the form fields in `<fieldset>` defined in variable `p_html`.

The commands of the Form utility are quiet straight-forward and easy to use. We also used the `radioset` and `checkset` command in the form in Figure 2.1 to show the syntax is just the same as those listed above. The usage is as below;

```
[<radioset> myname id mysex {Male male Female female}]
```

To add a legend to your form, you can use the `<legend>` command as in;

```
<legend> name {attrs} content
```

The `<legend>` command defines a caption for the `<fieldset>` element or command in this instance.

In the lines of code starting with the comments `# construct the form using a variety...` we used the `<hidden>`, `<text>`, `<password>`, `<radioset>`, `<select>` and `<submit>` commands of the Form utility. There are many more like `<file>`, `<image>`, `<textarea>`, `<button>`, `<reset>` etc. For more details, refer to the package in `/$install_dir/Utilities/Form.tcl`

Using jQ

If jQuery is “*write less, do more*”. That is exactly what jQ is in Wub. You do not have to know jQuery to use jQ. In a matter of minutes you will be creating fancy pages with tabs, accordions etc. with little effort.

JQ is implemented as a domain, to start using it, uncomment the lines that start with /jquery/ in site.config and you're ready to go.

The theme must also be started in your code before applying/using them. Here is the syntax and how you will use jQ domain to create a tabbed page;

```
jQ plugin_name req_dict #id
set r [jQ theme $r start]
set r [jQ tabs $r #tab123]
```

To pass jQ plugin options, you add them after the element id (#tab123) as a straight list as below;

```
set r [jQ tabs $r #tab123 active 1]
```

In our demonstration, we created tabs that contain an accordion in the first, a form in the second and text in the third tab.

First we will take a look at the page and then the code that created it. Note the page header is cut off in the images that follow.

Figure 2.3 jQ tabs with jQ accordion

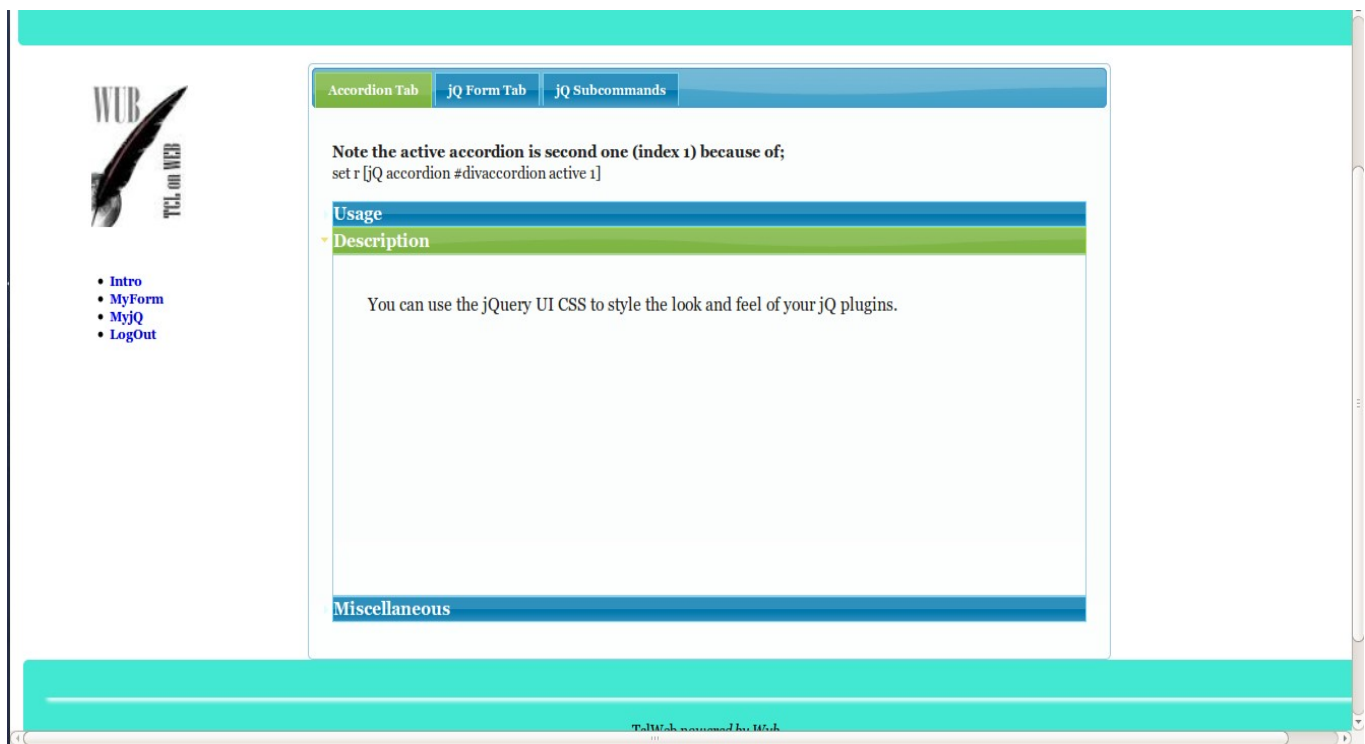


Fig 2.4 jQ tab with jQ Form

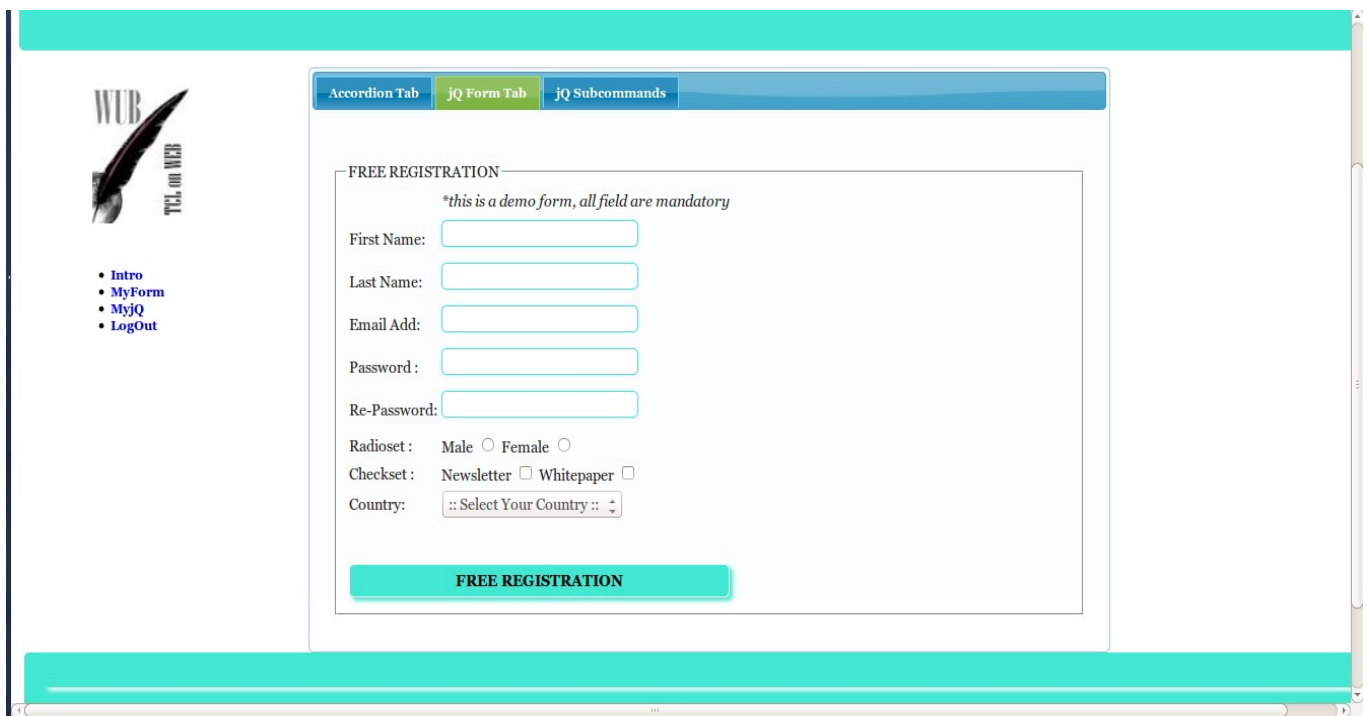
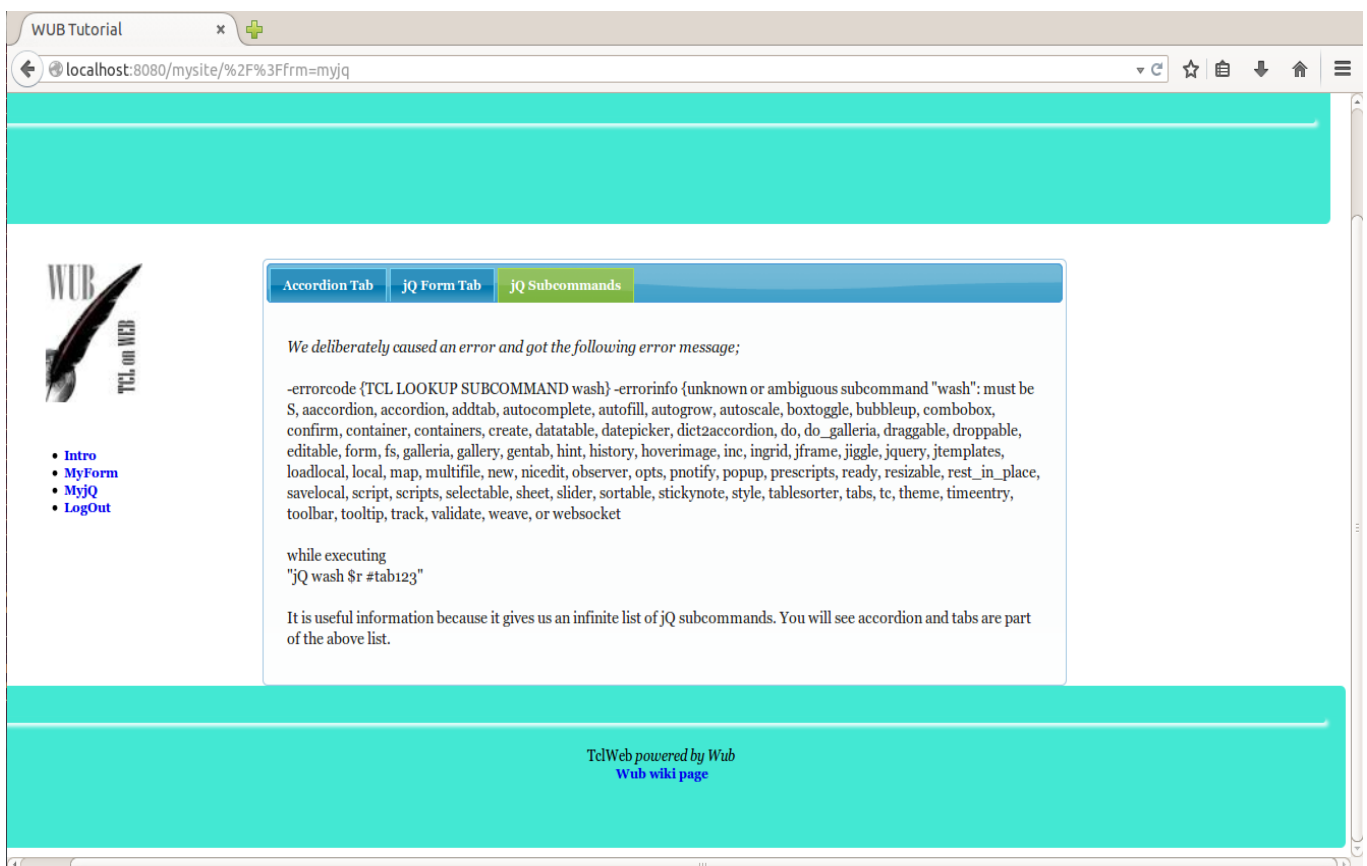


Fig 2.5 jQ tab with plain text



Prac 2.2 showing procedure in which the above page is constructed;

```
proc wtMyjQuery { r args } {

    # build the accordion ui
    set accUsage [<p> "
    1. The Nub is already configured in site.config. Uncomment the line if not already commented out to look like;
    [<br>]
    /jquery/ \{[<br>]
        domain JQ[<br>]
    \}
    [<br>]
    2. construct h3 and div tags wrapped in a div with an id[<br>]
    div id myacc h3 myhead div mycontent
    [<br>][<br>]
    3. set r \[jq accordion #id\]
    [<br>][<b> or][<br>]
    3.1 set r \[jq accordion #id option value option value option ...]\]
    [<br>]
"}

    set accDesc [<p> "
    Accordion two (index 1) is the current active pane[<br>][<br>]
    You can use the jQuery UI CSS to style the look and feel of your jq plugins.
    [<br>][<br>]
"}

    set tab3 [<p> "
    [<i> {We deliberately caused an error and got the following error message;}][<br>]
    [<br>]
    -errorcode \{TCL LOOKUP SUBCOMMAND wash\} -errorinfo \{unknown or ambiguous subcommand \"wash\": must be S,
    aaccordion, accordion, addtab, autocomplete,
    autofill, autogrow, autoscale, boxtoggle, bubbleup, combobox, confirm, container, containers, create,
    datatable, datepicker, dict2accordion, do,
    do_galleria, draggable, droppable, editable, form, fs, galleria, gallery, gentab, hint, history,
    hoverimage, inc, ingrid, jframe, jiggle, jquery,
    jtemplates, loadlocal, local, map, multifile, new, nicedit, observer, opts, pnotify, popup,
    prescripts, ready, resizable, rest_in_place, savelocal,
    script, scripts, selectable, sheet, slider, sortable, stickynote, style, tableserter, tabs, tc,
    theme, timeentry, toolbar, tooltip, track, validate,
    weave, or websocket
    [<br>][<br>]
    \while executing[<br>]
    \"jq wash \$r #tab123\"
    [<br>][<br>]
    It is useful information because it gives us an infinite list of jq subcommands. You will see accordion and
    tabs are part of the above list.
"}

    # construct a form to be made a jq form
    set wubform [<tr> [<td>[<td colspan 2 [<i> {*this is a demo form, all field are mandatory}]] ]
    append wubform [<tr> [<td colspan 2 [<hidden> hideme id hideme]] ]
    append wubform [<tr> [<td> "First Name:"][<td> [<text> fname id fname class frmfld title {First Name}]] ]
    append wubform [<tr> [<td> "Last Name:"][<td> [<text> lname id lname class frmfld title {Last Name}]] ]
    append wubform [<tr> [<td> "Email Add:"][<td> [<text> email id email class frmfld title {Email Address}]] ]
    append wubform [<tr> [<td> "Password :"]][<td> [<password> passwd id passwd class frmfld title {Password}]] ]
    append wubform [<tr> [<td> "Re-Password:"][<td> [<password> repeat id repeat class frmfld title {Retype Password}]] ]
    append wubform [<tr> [<td> "Radioset :"]][<td> [<radioset> mysex id mysex {Male male Female female}]] ]
    append wubform [<tr> [<td> "Checkset :"]][<td> [<checkboxset> mysurvey id mysurvey {Newsletter inews Whitepaper wpaper}]] ]
    append wubform [<tr> [<td> "Country:"][<td> [wtGetCountries]] ]
    append wubform [<tr> [<td colspan 2 [<br>][<br>][<submit> subReg id subReg class frmbutton {FREE REGISTRATION}]] ]
    set wubform [<table> $wubform]

    # build the form and insert the fields contained in variable 'wubform'
    set wubform [<br>][<form> frmjq id frmjq name frmjq action [Query::encode "?/reg"] method POST {
        [<fieldset> { [<legend> { FREE REGISTRATION }} $wubform }}
    ]}

    set wubform [<div> $wubform]

    # this section constructs the accordion
    set accordion [<h3> [<b> Usage]]
    append accordion [<div> $accUsage]
    append accordion [<h3> [<b> Description]]
    append accordion [<div> $accDesc]
    append accordion [<h3> [<b> Miscellaneous]]
    append accordion [<div> [<i> {Nothing to add, reserved for later use}]]
    set accordion [<div> id divaccordion $accordion]

    # the tabs plugin requires an ordered or unordered list with link to div elements
    set jqTab [<li> [<a href #tab1 {Accordion Tab}]]
    append jqTab [<li> [<a href #tab2 {jq Form Tab}]]
    append jqTab [<li> [<a href #tab3 {jq Subcommands}]]
    set jqTab [<ul> $jqTab]

    set wubAcc [<b> {Note the active accordion is second one (index 1) because of;}][<br>]
    append wubAcc "set r \[jq accordion #divaccordion active 1\"
    set wubAcc $wubAcc[<br>]$accordion

    # construct the contents of the all three tabs
    append jqTab [<div> id tab1 [<p> $wubAcc]]
    append jqTab [<div> id tab2 [<p> $wubform] ]
    append jqTab [<div> id tab3 [<p> $tab3] ]

    return [<br>][<br>][<div> id tab123 $jqTab]
}
```

The proc executed by clicking **MyjQ** menu item is proc wtMyJQuery above. It is called in the root proc rather than posting to the proc directly. The page itself contains useful information on jQ like the complete list of jQ plugins. Once your html elements have been built like in our proc above, you will apply jQ to your dict as was demonstrated earlier;

```
set r [jq plugin_name $r #id]
```

This concludes our discussion of jQ. The complete site code is distributed with this document to facilitate your playing around.

